EECS 280

Maps, The `auto` Keyword and Range-Based For Loops

4/4/2022

Example: Set Using a BST

```
template <typename T>
class BSTSet {
public:
  void insert(const T &v) {
    if (!elts.contains(v)) {
      elts.insert(v);
    }
  }

  bool contains(const T &v) const {
    return elts.contains(v);
  }

  int size() const {
    return elts.size();
  }

private:
  BinarySearchTree<T> elts;
};
```

This is the "has-a" pattern. The data representation for the BSTSet is primarily just a BinarySearchTree, which does all the work behind the scenes.

We saw this pattern in project 4 with the Stack that had a List internally.

You'll also use it in project 5 to implement the Map using a BinarySearchTree.

Set Efficiency

- How efficient is each operation?

	UnsortedSet	SortedSet	BSTSet
insert	O(n)	O(n)	O(log n)
remove	O(n)	O(n)	O(log n)
contains	O(n)	O(log n)	O(log n)
size	O(1)	O(1)	O(n)
constructor	O(1)	O(1)	O(1)

Need to check if set contains item.

Average time.

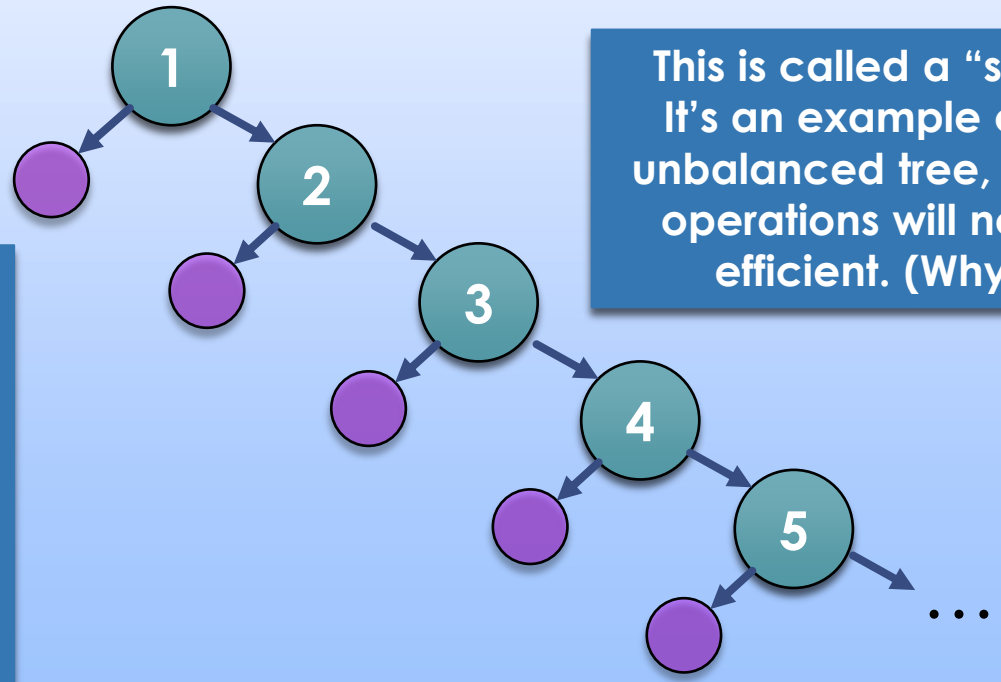
Can be made O(1).

A Problem

- Let's say we insert a sequence of numbers into our BST:

  1, 2, 3, 4, 5, 6, 7, 8

- What does the resulting tree look like?



This is called a "stick". It's an example of an unbalanced tree, where operations will not be efficient. (Why?)

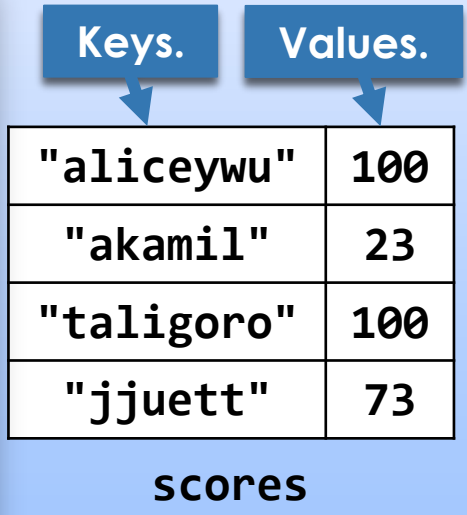
Your project 5 BST, and thus your `Map` ADT, will be susceptible to this problem. Don't worry about it.

For the top-level classifier application, just use the STL version `std::map`.

Maps

- A *map* is a data structure that associates keys with values
- The *key* is what we use to look up or insert an item
- The *value* is what is associated with the key

```
int main() {
  map<string, int> scores;
  scores["aliceywu"] = 100;
  scores["akamil"] = 23;
  scores["taligoro"] = 100;
  scores["jjuett"] = 73;
  cout << scores["akamil"] << endl;
  cout << scores["aliceywu"] << endl;
}
```

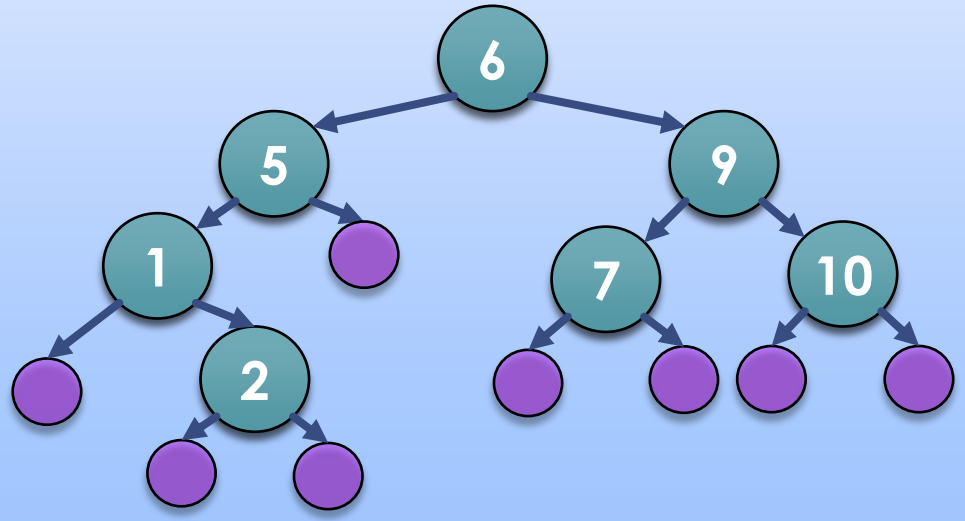


Review: Binary Search Trees (BSTs)

- A tree is a binary search tree if…
  - It is empty

  OR
  - The left and right subtrees are binary search trees.
  - All elements in any **left** subtree are **less** than the root.
  - All elements in any **right** subtree are **greater** than the root.

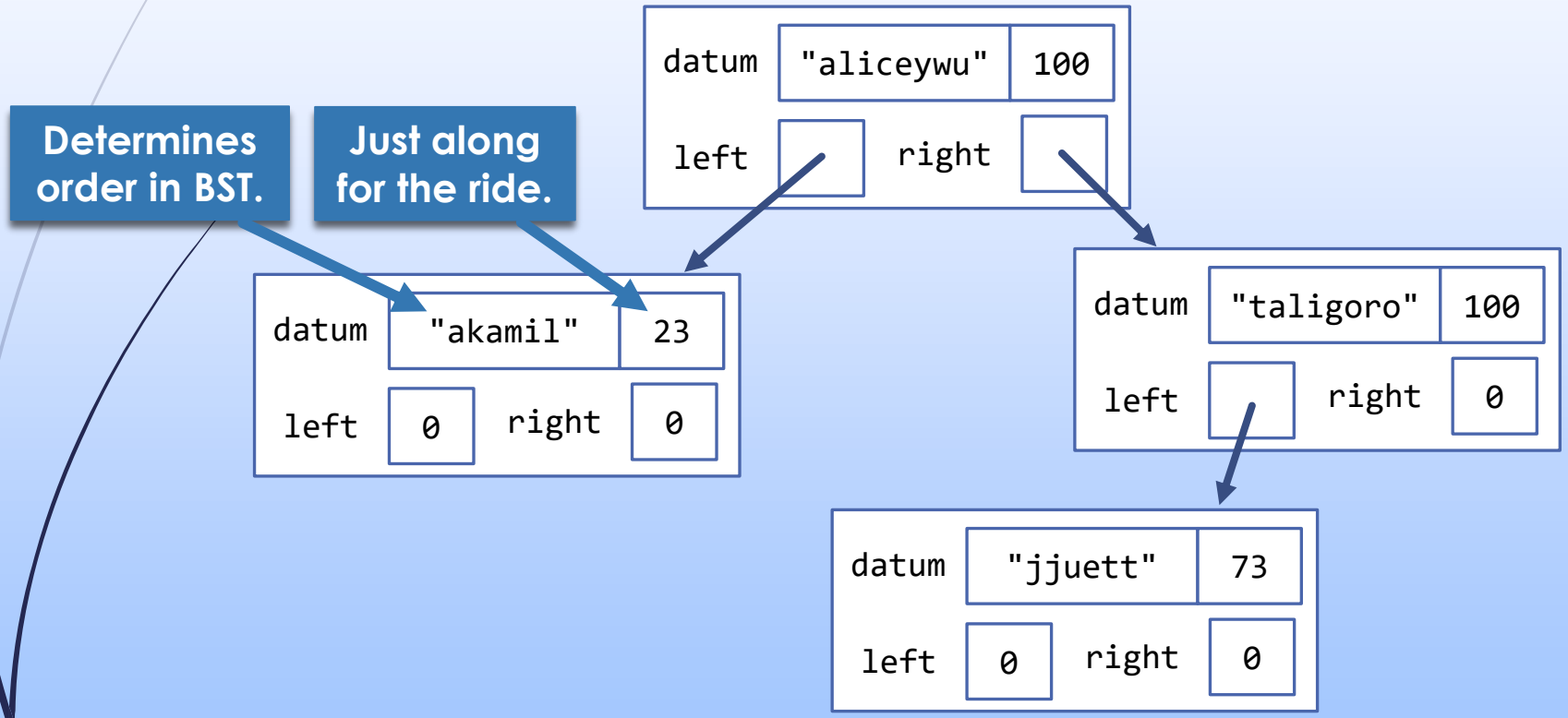
It is so called because searching for elements can be done efficiently.



Note: In the slides and on project 5, we make a simplifying assumption that there are no duplicates in our BSTs.

Representing a Map

- We can use a BST to store the data in a map



std::pair

- `std::pair` is an STL class template that can be used to represent a pair of objects.
- The template parameters determine the type of the first and second objects.
- For example:

```
std::pair<int, bool> p1;
p1.first = 5;
p1.second = false;

std::pair<string, int> p2;
p2.first = "hello";
p2.second = 4;
```

`std::pair` is essentially a `struct` with members called `first` and `second`.

Implementing a Map

```
template <typename Key_type, typename Value_type,
          typename Key_Compare>
class Map {
public:
  bool empty() const;
  size_t size() const;
  Value_type& operator[](const Key_type& k);

private:
  using Pair_type = std::pair<Key_type, Value_type>;

  class PairComp {
  public:
    bool operator()(...);
  };

  BinarySearchTree<Pair_type, PairComp> entries;
};
```

Type alias for convenience.

Custom comparator to order pairs.

New BST ADT that can take a custom comparator.

Map Functions

```
// EFFECTS : Searches this Map for an element with a key equivalent
//           to k and returns an Iterator to the associated value
//           if found, otherwise returns an end Iterator.
Iterator find(const Key_type& k) const;
```

Give me an iterator to <key,value> based on a key
If it's not there, give me an end iterator

```
// EFFECTS : Inserts the given element into this Map if the given key
//           is not already contained in the Map. If the key is
//           already in the Map, returns an iterator to the
//           corresponding existing element, along with the value
//           false. Otherwise, inserts the given element and returns
//           an iterator to the newly inserted element, along with
//           the value true.
std::pair<Iterator, bool> insert(const Pair_type &val);
```

Insert this pair of <key,value>
Give me a fancy return with info about what was inserted

Map Functions

```
// EFFECTS : Returns a reference to the mapped value for the given
//           key. If k matches the key of an element in the
//           container, the function returns a reference to its
//           mapped value. If k does not match the key of any
//           element in the container, the function inserts a new
//           element with that key and a value-initialized mapped
//           value and returns a reference to the mapped value.
//           Note: value-initialization for numeric types guarantees the
//           value will be 0 (rather than memory junk).
//
// HINT:     In the case the key was not found, and you must insert a
//           new element, use the expression {k, Value_type()} to create
//           that element. This ensures the proper value-initialization
is done.
//
// HINT: http://www.cplusplus.com/reference/map/map/operator[]/
Value_type& operator[](const Key_type& k);
```

Give me a reference to the value for this key.
If it wasn't there, add a default placeholder with 0.

Type Deduction with auto

- The auto keyword tells the compiler to automatically deduce the type of a variable.

```
template <typename T>
class List {
public:
  Iterator begin() { return Iterator(first); }
  ...
};
```

Returns an object of type List<int>::Iterator

```
int main() {
  List<int> lst;
  for (auto it = lst.begin(); it != lst.end(); ++it) {
    *it = 42;  // fill with 42
  }
}
```

Deduced to have type List<int>::Iterator

Range-Based For Loop

- A *range-based for loop* is a special syntax for iterating over a sequence.
- It automatically:
  - Calls `begin()` and `end()` on a sequence to get start and end iterators.
  - Initializes the given variable in each iteration by dereferencing the start iterator.
  - Increments the start iterator after each iteration.

```
vector<int> vec(5);
for (int item : vec) {
  cout << item << endl;
}
```

```
vector<int> vec(5);
auto it = vec.begin();
auto end_it = vec.end();
for (; it != end_it; ++it) {
  int item = *it;
  cout << item << endl;
}
```

Exercise: Range-Based For Loop

- What does the following code print?

```
int main() {
  vector<int> vec(5);

  for (int item : vec) {
    item = 42;
  }

  for (int item : vec) {
    cout << item << endl;
  }
}
```

Solution: Range-Based For Loop

- What does the following code print?

```
vector<int> vec(5);
for (int item : vec) {
 item = 42;
}

for (int item : vec) {
  cout << item << endl;
}
```

Not the object in the sequence.

Prints junk values.

```
vector<int> vec(5);
auto it = vec.begin();
auto end_it = vec.end();
for (; it != end_it; ++it) {
  int item = *it;
  item = 42;
}

auto it2 = vec.begin();
auto end_it2 = vec.end();
for (; it2 != end_it2; ++it2) {
  int item = *it2;
  cout << item << endl;
}
```

Solution: Range-Based For Loop

- We can fix the code by declaring a reference.

```
vector<int> vec(5);
for (int &item : vec) {
 item = 42;
}

for (int item : vec) {
  cout << item << endl;
}
```

Aliases the object in the sequence.

Prints 42 five times.

```
vector<int> vec(5);
auto it = vec.begin();
auto end_it = vec.end();
for (; it != end_it; ++it) {
  int &item = *it;
  item = 42;
}

auto it2 = vec.begin();
auto end_it2 = vec.end();
for (; it2 != end_it2; ++it2) {
  int item = *it2;
  cout << item << endl;
}
```

Range-Based For Loop with auto

- We can use the **auto** keyword to deduce the element type in a range-based for loop.

```
vector<int> vec(5);
for (auto &item : vec) {
 item = 42;
}

for (auto item : vec) {
  cout << item << endl;
}
```

Aliases the object in the sequence.

Does not alias the object in the sequence.

```
vector<int> vec(5);
auto it = vec.begin();
auto end_it = vec.end();
for (; it != end_it; ++it) {
  int &item = *it;
  item = 42;
}

auto it2 = vec.begin();
auto end_it2 = vec.end();
for (; it2 != end_it2; ++it2) {
  int item = *it2;
  cout << item << endl;
}
```

Using a Map: Word Counts

- Let's say we have a vector of words and we want to count how many times each word occurs…

```
void printWordCounts(const vector<string> &words) {
  std::map<string, int> wordCounts;

  // Each time a word is seen, add 1 to its entry in
  // the map. If it wasn't there, make a 0
  // placeholder and then immediately add 1 to that
  for (const auto &word : words) {
    wordCounts[word] += 1;
  }

  // Print out results by iterating through the map
  for (const auto &kv : wordCounts) {
    const auto &word = kv.first;
    const auto &count = kv.second;
    cout << word << "occurred "
         << count << " times." << endl;
  }
}
```